1. Giải thích code

Đọc dữ liệu từ file csv

df = pd.read\_csv('sinh\_vien\_data.csv')

df

Vẽ biểu đồ để xem phân phối dữ liệu của từng cột

# Tạo figure và vẽ histogram

plt.figure(figsize=(15,6))

counts, bins, patches = plt.hist(df["Điểm 1"], bins=10, color='#C0EBA6', edgecolor='black') -> Dùng biểu đồ hist để xem phân phối điểm

# Đặt tiêu đề và nhãn trục

plt.title('Phân phối điểm 1')

plt.xlabel('Điểm')

plt.ylabel('Tần suất')

# Thêm nhãn số liệu lên các cột của biểu đồ

for count, patch in zip(counts, patches):

    plt.text(patch.get\_x() + patch.get\_width() / 2,  # Vị trí x của nhãn (giữa cột)

             count,                                 # Vị trí y của nhãn (tần suất)

             str(int(count)),                       # Nội dung nhãn (tần suất dưới dạng số nguyên)

             ha='center',                           # Canh giữa

             va='bottom')                           # Đặt nhãn phía trên cột

# Hiển thị biểu đồ

plt.show()

Chọn đặc trưng và nhãn

Với X, chọn đặc trưng là điểm thứ 1, điểm thứ 2, điểm thứ 3

Nhãn là điểm thi

X = df.drop(columns=["Họ và tên", "Điểm thi"])

y = df["Điểm thi"]

Chia tập train – test với tỉ lệ 80 – 20

X\_train, X\_test, y\_train, y\_test = train\_test\_split(X, y, test\_size = 0.2, random\_state=42)

Chuẩn hóa dữ liệu của tập đặc trưng bằng MinMax

scaler = MinMaxScaler()

X\_train = scaler.fit\_transform(X\_train)

X\_test = scaler.transform(X\_test)

Huấn luyện mô hình Linear Regression và tính các chỉ số MSE, MAE, MAPE, RSME

model = LinearRegression()

model.fit(X\_train, y\_train)

y\_pred = model.predict(X\_test)

mse\_lr = mean\_squared\_error(y\_test, y\_pred)

mae\_lr = mean\_absolute\_error(y\_test, y\_pred)

mape\_lr = mean\_absolute\_percentage\_error(y\_test, y\_pred)

rmse\_lr = root\_mean\_squared\_error(y\_test, y\_pred)

**Xây mô hình CNN**

import tensorflow as tf

class CNNModel(tf.Module):

    def \_\_init\_\_(self):

        super(CNNModel, self).\_\_init\_\_()

        self.conv1 = tf.Variable(tf.random.normal([1, 1, 1, 32]), name='conv1\_weights', dtype=tf.float32)  # Lớp Conv2D với 32 filters

        self.dense1 = tf.Variable(tf.random.normal([96, 48]), name='dense1\_weights', dtype=tf.float32)  # Fully connected layer 1

        self.dense2 = tf.Variable(tf.random.normal([48, 1]), name='dense2\_weights', dtype=tf.float32)   # Fully connected layer 2

    def \_\_call\_\_(self, x):

        x = tf.reshape(x, [-1, 3, 1, 1])  # Reshape input thành [batch\_size, height=2, width=1, channels=1]

        conv = tf.nn.conv2d(x, self.conv1, strides=[1, 1, 1, 1], padding="VALID")

        conv = tf.nn.relu(conv)  # Activation ReLU

        conv\_flat = tf.reshape(conv, [-1, 96])  # Flatten

        dense1\_out = tf.matmul(conv\_flat, self.dense1)

        dense1\_out = tf.nn.relu(dense1\_out)  # Activation ReLU

        output = tf.matmul(dense1\_out, self.dense2)  # Output layer

        return output

# Khởi tạo mô hình

model = CNNModel()

# Hàm mất mát (loss function)

loss\_fn = tf.losses.MeanSquaredError()

# Optimizer

optimizer = tf.optimizers.Adam(learning\_rate=0.001)

# Hàm huấn luyện một batch

@tf.function

def train\_step(X, y):

    with tf.GradientTape() as tape:

        predictions = model(X)

        loss = loss\_fn(y, predictions)

    gradients = tape.gradient(loss, model.trainable\_variables)

    optimizer.apply\_gradients(zip(gradients, model.trainable\_variables))

    return loss

# Giả định đã có dữ liệu X\_train, y\_train, X\_test, y\_test

X\_train\_dl = X\_train.astype(np.float32)

y\_train\_dl = y\_train.astype(np.float32)

X\_test\_dl = X\_test.astype(np.float32)

y\_test\_dl = y\_test.astype(np.float32)

# Lưu giá trị loss

train\_loss\_values = []

test\_loss\_values = []

# Huấn luyện mô hình

epochs = 500

batch\_size = 32

for epoch in range(epochs):

    epoch\_loss = 0

    # Huấn luyện

    for i in range(0, len(X\_train\_dl), batch\_size):

        X\_batch = X\_train\_dl[i:i + batch\_size]

        y\_batch = y\_train\_dl[i:i + batch\_size]

        loss = train\_step(X\_batch, y\_batch)

        epoch\_loss += loss.numpy()

    avg\_train\_loss = epoch\_loss / (len(X\_train\_dl) // batch\_size)

    train\_loss\_values.append(avg\_train\_loss)

    # Tính loss của tập test

    test\_predictions = model(X\_test\_dl)

    test\_loss = loss\_fn(y\_test\_dl, test\_predictions)

    avg\_test\_loss = test\_loss.numpy()

    test\_loss\_values.append(avg\_test\_loss)

    print(f'Epoch {epoch + 1}/{epochs}, Train Loss: {avg\_train\_loss:.4f}, Test Loss: {avg\_test\_loss:.4f}')

Mô hình CNN gồm 5 lớp: Conv2D -> ReLU -> Matmul -> ReLU -> Matmul

Sử dụng hàm mất mát là MSE và hàm tối ưu là Adam với learning rate = 0.001

Train\_step là hàm dung để huấn luyện mô hình qua từng epoch. Tính toán sai số và tối ưu dần mô hình

Huấn luyện mô hình với 500 epochs, batch size là 32.

**Xây mô hình RNN**

import tensorflow as tf

# Giả định đã có dữ liệu X\_train, y\_train, X\_test, y\_test

# Dưới đây là mô hình RNN được định nghĩa như trên

class SimpleRNN:

    def \_\_init\_\_(self, input\_size, hidden\_size):

        self.hidden\_size = hidden\_size

        self.Wxh = tf.Variable(tf.random.normal([input\_size, hidden\_size]), name='Wxh')

        self.Whh = tf.Variable(tf.random.normal([hidden\_size, hidden\_size]), name='Whh')

        self.bh = tf.Variable(tf.zeros([hidden\_size]), name='bh')

        self.Why = tf.Variable(tf.random.normal([hidden\_size, 1]), name='Why')

        self.by = tf.Variable(tf.zeros([1]), name='by')

    def step(self, x, h):

        h = tf.tanh(tf.matmul(x, self.Wxh) + tf.matmul(h, self.Whh) + self.bh)

        y = tf.matmul(h, self.Why) + self.by

        return y, h

    def forward(self, x):

        h = tf.zeros([x.shape[0], self.hidden\_size])  # Initialize hidden state

        for t in range(x.shape[1]):  # Loop over time steps

            y, h = self.step(x[:, t, :], h)

        return y

    @property

    def trainable\_variables(self):

        return [self.Wxh, self.Whh, self.bh, self.Why, self.by]

# Tạo mô hình

model = SimpleRNN(input\_size=3, hidden\_size=64)

# Tạo optimizer

optimizer = tf.optimizers.Adam()

# Hàm tính loss

loss\_fn = tf.losses.MeanSquaredError()

@tf.function

def train\_step(x, y):

    with tf.GradientTape() as tape:

        predictions = model.forward(x)

        loss = loss\_fn(y, predictions)

    gradients = tape.gradient(loss, model.trainable\_variables)

    optimizer.apply\_gradients(zip(gradients, model.trainable\_variables))

    return loss

# Hàm tính loss trên tập test

@tf.function

def test\_loss(x, y):

    predictions = model.forward(x)

    loss = loss\_fn(y, predictions)

    return loss

# Chuyển đổi dữ liệu sang định dạng phù hợp

X\_train\_dl = X\_train.astype(np.float32).reshape(-1, 1, 3)

y\_train\_dl = y\_train.values.astype(np.float32).reshape(-1, 1, 1)

X\_test\_dl = X\_test.astype(np.float32).reshape(-1, 1, 3)

y\_test\_dl = y\_test.values.astype(np.float32).reshape(-1, 1, 1)

# Định nghĩa số epoch và batch size

epochs = 500

batch\_size = 32

# List lưu loss để vẽ biểu đồ

train\_losses = []

test\_losses = []

# Vòng lặp huấn luyện

for epoch in range(epochs):

    # Huấn luyện trên tập train

    for i in range(0, len(X\_train\_dl), batch\_size):

        X\_batch = X\_train\_dl[i:i + batch\_size]

        y\_batch = y\_train\_dl[i:i + batch\_size]

        train\_loss = train\_step(X\_batch, y\_batch)

    # Tính loss trên tập test sau mỗi epoch

    test\_loss\_value = test\_loss(X\_test\_dl, y\_test\_dl)

    # Lưu lại loss để vẽ biểu đồ

    train\_losses.append(train\_loss.numpy())

    test\_losses.append(test\_loss\_value.numpy())

    print(f'Epoch {epoch + 1}/{epochs}, Train Loss: {train\_loss.numpy():.4f}, Test Loss: {test\_loss\_value.numpy():.4f}')

1. Định nghĩa lớp SimpleRNN

\_\_init\_\_(self, input\_size, hidden\_sizes): Khởi tạo mô hình RNN với input\_size là kích thước đầu vào và hidden\_sizes là danh sách các đơn vị ẩn của mỗi lớp RNN.

Wxhs: Trọng số kết nối giữa đầu vào và trạng thái ẩn của từng lớp.

Whhs: Trọng số kết nối giữa các trạng thái ẩn của cùng một lớp.

bhs: Bias cho từng lớp ẩn.

Why và by: Trọng số và bias cho lớp đầu ra (output layer).

2. step(self, x, hs): Tính toán trạng thái ẩn tại mỗi bước thời gian

x: Đầu vào hiện tại.

hs: Trạng thái ẩn của từng lớp từ bước trước đó.

Tính toán trạng thái ẩn cho từng lớp sử dụng hàm tanh và trả về danh sách trạng thái ẩn new\_hs.

3. forward(self, x): Dự đoán đầu ra cho toàn bộ chuỗi thời gian

hs: Khởi tạo trạng thái ẩn cho từng lớp bằng tf.zeros.

Lặp qua từng bước thời gian t để cập nhật hs.

y: Đầu ra cuối cùng được tính toán từ trạng thái ẩn của lớp cuối cùng.

4. trainable\_variables: Trả về danh sách các biến có thể train

Bao gồm trọng số và bias của tất cả các lớp trong mô hình.

5. Khởi tạo mô hình

model = SimpleRNN(input\_size=3, hidden\_sizes=[64, 64, 64, 64, 64]): Tạo mô hình RNN với đầu vào có kích thước 3 và 5 lớp ẩn, mỗi lớp có 64 đơn vị ẩn.

6. Khởi tạo optimizer và loss\_fn

optimizer = tf.optimizers.Adam(): Sử dụng Adam optimizer.

loss\_fn = tf.losses.MeanSquaredError(): Sử dụng Mean Squared Error (MSE) làm hàm tính loss.

7. train\_step(x, y): Hàm train cho từng batch

Dùng GradientTape để theo dõi các biến trong mô hình.

predictions: Dự đoán đầu ra từ mô hình.

loss: Tính loss giữa y thật và predictions.

gradients: Tính toán gradient của loss với từng biến trong mô hình.

optimizer.apply\_gradients: Cập nhật các biến dựa trên gradient.

8. test\_loss(x, y): Hàm tính loss trên tập test

Tương tự train\_step nhưng không tính gradient.

9. Chuyển đổi dữ liệu

X\_train\_dl và X\_test\_dl: Chuyển đổi dữ liệu đầu vào thành định dạng [số mẫu, 1, kích thước đầu vào] để phù hợp với RNN.

y\_train\_dl và y\_test\_dl: Chuyển đổi dữ liệu nhãn thành định dạng [số mẫu, 1, 1].

10. Vòng lặp huấn luyện

for epoch in range(epochs): Vòng lặp qua từng epoch.

for i in range(0, len(X\_train\_dl), batch\_size): Chia dữ liệu thành từng batch.

train\_loss = train\_step(X\_batch, y\_batch): Huấn luyện trên từng batch.

test\_loss\_value = test\_loss(X\_test\_dl, y\_test\_dl): Tính loss trên tập test sau mỗi epoch.

train\_losses và test\_losses: Lưu lại các giá trị loss để theo dõi và vẽ biểu đồ.

11. In kết quả mỗi epoch

print(...): Hiển thị loss của tập train và test sau mỗi epoch.

Mô hình được xây dựng theo từng lớp như sau:

Layer 1: Input layer

layers.Input(shape=input\_shape): Khởi tạo đầu vào với kích thước input\_shape là (3, 1). Đây là lớp đầu vào giúp mô hình biết trước kích thước dữ liệu.

Layer 2: Convolutional layer

layers.Conv1D(32, kernel\_size=1, activation='relu'): Lớp tích chập 1 chiều (1D Convolutional Layer) với 32 filters, kích thước kernel là 1 và hàm kích hoạt ReLU. Lớp này sẽ trích xuất các đặc trưng của dữ liệu đầu vào.

Layer 3: Convolutional layer

layers.Conv1D(64, kernel\_size=1, activation='relu'): Lớp tích chập 1 chiều với 64 filters, kích thước kernel là 1 và hàm kích hoạt ReLU. Lớp này tiếp tục trích xuất thêm các đặc trưng từ đầu ra của lớp tích chập trước.

Layer 4: Flatten layer

layers.Flatten(): Lớp này chuyển đổi đầu ra của các lớp trước (dạng 2D) thành một vector 1D để làm đầu vào cho lớp fully connected.

Layer 5: Fully connected layer

**Mô hình CNN Keras**

from tensorflow.keras import layers, models

input\_shape = (3, 1)  # 2 đặc trưng và 1 kênh (channel)

# Xây dựng mô hình CNN

model = models.Sequential([

    layers.Input(shape=input\_shape),                      # Layer 1: Input layer

    layers.Conv1D(32, kernel\_size=1, activation='relu'),   # Layer 2: Convolutional layer

    layers.Conv1D(64, kernel\_size=1, activation='relu'),   # Layer 3: Convolutional layer

    layers.Flatten(),                                     # Layer 4: Flatten layer

    layers.Dense(64, activation='relu'),                   # Layer 5: Fully connected layer

    layers.Dense(1)                                        # Output layer: Dự đoán điểm cuối cùng

])

# Compile mô hình

model.compile(optimizer='adam', loss='mean\_squared\_error')

# In tóm tắt mô hình

model.summary()

layers.Dense(64, activation='relu'): Lớp fully connected với 64 đơn vị ẩn và hàm kích hoạt ReLU. Lớp này kết nối toàn bộ các đầu vào từ lớp Flatten để học các đặc trưng phức tạp hơn.

Output layer

layers.Dense(1): Lớp đầu ra với 1 đơn vị, không có hàm kích hoạt (tức là hàm tuyến tính). Lớp này đưa ra kết quả dự đoán cuối cùng, thường được sử dụng cho bài toán hồi quy.

optimizer='adam': Sử dụng Adam optimizer để cập nhật trọng số trong quá trình huấn luyện.

loss='mean\_squared\_error': Sử dụng hàm loss Mean Squared Error (MSE) để đo lường sự khác biệt giữa dự đoán và giá trị thực tế.

model.fit(X\_train, y\_train, epochs=25, batch\_size=32, validation\_data=(X\_test, y\_test))

Huấn luyện mô hình với tập train, epochs = 25 và batch size là 32

**Xây dựng mô hình RNN Keras**

model = models.Sequential([

    layers.Input(shape=(3, 1)),                     # Layer 1: Input layer

    layers.SimpleRNN(32, activation='relu', return\_sequences=True),  # Layer 2: RNN layer (trả về toàn bộ chuỗi)

    layers.SimpleRNN(64, activation='relu'),        # Layer 3: RNN layer

    layers.Dense(32, activation='relu'),            # Layer 4: Fully connected layer

    layers.Dense(1)                                 # Output layer: Dự đoán điểm cuối cùng

])

# Compile mô hình

model.compile(optimizer='adam', loss='mean\_squared\_error')

# In tóm tắt mô hình

model.summary()

Mô hình bao gồm 5 lớp:

Layer 1: Input layer

layers.Input(shape=(3, 1)): Khởi tạo lớp đầu vào với kích thước shape=(3, 1). Đầu vào là chuỗi có 3 bước thời gian (3 timesteps) và mỗi bước có 1 đặc trưng (1 feature).

Layer 2: RNN layer

layers.SimpleRNN(32, activation='relu', return\_sequences=True):

Đây là lớp RNN với 32 đơn vị ẩn và hàm kích hoạt ReLU.

return\_sequences=True: Tùy chọn này cho phép lớp RNN trả về toàn bộ chuỗi trạng thái ẩn (hidden states) của từng bước thời gian. Điều này hữu ích khi muốn nối tiếp lớp này với một lớp RNN khác (như trong mô hình này).

Layer 3: RNN layer

layers.SimpleRNN(64, activation='relu'):

Lớp RNN với 64 đơn vị ẩn và hàm kích hoạt ReLU.

Không có return\_sequences=True, do đó lớp này chỉ trả về trạng thái ẩn cuối cùng của chuỗi thời gian. Trạng thái ẩn cuối cùng này sẽ được truyền vào lớp kế tiếp.

Layer 4: Fully connected layer

layers.Dense(32, activation='relu'):

Lớp fully connected với 32 đơn vị ẩn và hàm kích hoạt ReLU.

Lớp này kết nối toàn bộ các đầu vào từ lớp RNN để học các đặc trưng phức tạp hơn.

Output layer

layers.Dense(1):

Lớp đầu ra với 1 đơn vị, không có hàm kích hoạt (hàm tuyến tính).

Đầu ra này thường được sử dụng cho các bài toán hồi quy, cho phép mô hình dự đoán một giá trị liên tục.

optimizer='adam': Sử dụng Adam optimizer để cập nhật trọng số trong quá trình huấn luyện.

loss='mean\_squared\_error': Sử dụng hàm loss Mean Squared Error (MSE) để đo lường sự khác biệt giữa giá trị dự đoán và giá trị thực tế. Hàm loss này phù hợp với các bài toán hồi quy.

model.fit(X\_train, y\_train, epochs=25, batch\_size=32, validation\_data=(X\_test, y\_test))

Huấn luyện mô hình với tập train, 25 epochs với batch size là 32

So sánh các mô hình với MSE, MAE, RSME, MAPE



1. Mean Squared Error (MSE):

* Linear Regression và CNN có MSE tương đối thấp (0.49 và 0.48).
* RNN có MSE rất cao (6.31), cho thấy mô hình này hoạt động không hiệu quả.
* CNN\_Keras và RNN\_Keras cải thiện hơn so với RNN thuần túy, đạt MSE là 0.53 và 0.49.

2. Mean Absolute Error (MAE):

* Linear Regression và CNN có MAE thấp và tương đương nhau (0.57).
* RNN có MAE cao nhất (2.12).
* CNN\_Keras và RNN\_Keras cũng cải thiện so với RNN, đạt lần lượt 0.58 và 0.56.

3. Mean Absolute Percentage Error (MAPE):

* RNN có MAPE rất lớn (1.43 x 10^14), cho thấy mô hình này hoạt động kém nhất.
* Các mô hình còn lại (Linear Regression, CNN, CNN\_Keras, và RNN\_Keras) đều có MAPE thấp hơn rất nhiều.

4. Root Mean Squared Error (RMSE):

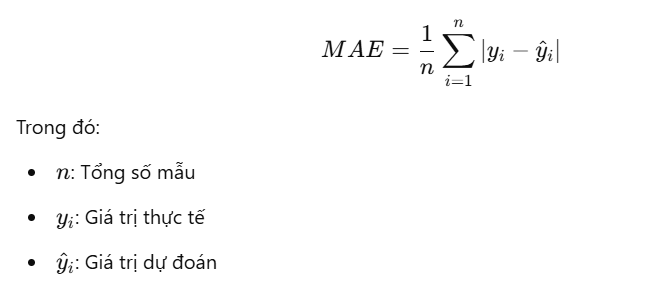
* Linear Regression và CNN có RMSE gần nhau (0.70).
* RNN có RMSE cao (2.51), cho thấy sai số dự đoán lớn.
* CNN\_Keras và RNN\_Keras có RMSE tốt hơn (0.72 và 0.70).

Câu 2:

**1. Mean Absolute Error (MAE) - Sai số trung bình tuyệt đối**

**Định nghĩa**: MAE đo lường sai số trung bình giữa giá trị dự đoán và giá trị thực tế. Nó là giá trị trung bình của độ lệch tuyệt đối giữa các giá trị dự đoán và thực tế.

* **Công thức**:

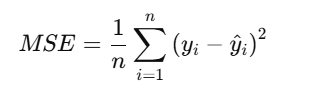


* **Ý nghĩa**: MAE cho biết độ lệch trung bình mà mô hình tạo ra giữa các dự đoán và thực tế mà không quan tâm đến việc dự đoán cao hơn hay thấp hơn. MAE dễ hiểu và dễ giải thích, nhưng không nhạy cảm với các sai số lớn (outliers).
* **Đơn vị**: Đơn vị của MAE giống như đơn vị của biến đầu ra.

**2. Mean Squared Error (MSE) - Sai số bình phương trung bình**

**Định nghĩa**: MSE đo lường độ lớn trung bình của sai số bình phương giữa giá trị dự đoán và giá trị thực tế.

* **Công thức**:

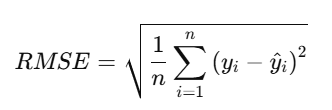


* **Ý nghĩa**: MSE nhấn mạnh các sai số lớn hơn, vì các sai số được bình phương trước khi tính trung bình. Nếu mô hình có những sai số lớn, MSE sẽ tăng rất nhanh. Điều này giúp MSE trở nên nhạy cảm hơn với các outlier.
* **Đơn vị**: Đơn vị của MSE là bình phương của đơn vị biến đầu ra, nên khó so sánh trực tiếp với các metric khác như MAE.

**3. Root Mean Squared Error (RMSE) - Căn bậc hai của MSE**

**Định nghĩa**: RMSE là căn bậc hai của MSE, giúp chuyển đổi lại đơn vị của MSE về cùng đơn vị với biến đầu ra.

* **Công thức**:

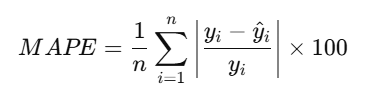


* **Ý nghĩa**: RMSE giữ tính chất nhạy cảm với các outlier giống như MSE, nhưng nó cho kết quả dễ so sánh và dễ hiểu hơn vì cùng đơn vị với biến đầu ra. RMSE thường được sử dụng để đánh giá mức độ khớp giữa mô hình và dữ liệu thực tế.
* **Đơn vị**: Đơn vị của RMSE giống như đơn vị của biến đầu ra.

**4. Mean Absolute Percentage Error (MAPE) - Sai số phần trăm trung bình tuyệt đối**

**Định nghĩa**: MAPE đo lường độ lệch trung bình theo tỷ lệ phần trăm giữa giá trị dự đoán và giá trị thực tế.

* **Công thức**:



* **Ý nghĩa**: MAPE cho biết sai số trung bình tính theo phần trăm, giúp hiểu rõ hơn về mức độ sai lệch tương đối so với giá trị thực tế. Tuy nhiên, MAPE không thể tính được khi giá trị thực tế yiy\_iyi​ bằng 0, và nó có thể bị lệch khi giá trị thực tế rất nhỏ (vì sẽ tạo ra phần trăm rất lớn).
* **Đơn vị**: Đơn vị của MAPE là phần trăm (%).

**So sánh các metrics**

* **MAE**: Dễ hiểu, không nhạy cảm với outlier, nhưng không phản ánh chính xác khi có sai số lớn.
* **MSE**: Nhạy cảm với outlier, thích hợp khi muốn đánh giá ảnh hưởng của các dự đoán sai lớn.
* **RMSE**: Kế thừa đặc điểm của MSE, nhưng dễ so sánh hơn do cùng đơn vị với biến đầu ra.
* **MAPE**: Biểu diễn sai số dưới dạng phần trăm, nhưng không phù hợp với dữ liệu có giá trị thực bằng 0 hoặc rất nhỏ.

**Khi nào sử dụng từng metrics?**

* **MAE**: Khi muốn một thước đo dễ hiểu và không nhạy cảm với outliers.
* **MSE/RMSE**: Khi muốn đánh giá mức độ ảnh hưởng của các dự đoán sai lệch lớn.
* **MAPE**: Khi cần so sánh mức độ sai lệch tương đối và dữ liệu không có giá trị thực bằng 0 hoặc quá nhỏ.